

**AN IMPLEMENTATION OF A CAMERA
TECHNIQUE TO OBTAIN
THREE-DIMENSIONAL (3D) VISION
INFORMATION FOR SIMPLE
ASSEMBLY TASKS**

NAGW-1333

by

Deepak Sood, Michael C. Repko, and Robert B. Kelley

Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering
Troy, New York 12180-3590

November, 1990

CIRSSE REPORT #76

An Implementation of a Camera Calibration Technique to obtain Three-Dimensional (3-D) Vision Information for Simple Assembly Tasks

Deepak Sood, Michael C. Repko and Robert B. Kelley

Center for Intelligent Robotic Systems for Space Exploration

Rensselaer Polytechnic Institute

Troy, New York 12180-3590

Fall 1990

Abstract

Camera calibration is the problem of determining the elements that govern the relationship of transformation between the 2-D image that a camera perceives and the 3-D information of the imaged object.

Camera calibration is important because it allows the determination of the relative rotation and translation between the lens centered coordinate system and an arbitrarily defined world coordinate system. For applications where

visual information is required by the robot to perform tasks, the cameras can be calibrated with respect to the robot base coordinate system.

After the camera is properly calibrated, its 2-D image coordinates can be translated into real world locations for the objects in its field of view. Thus, the 3D location of objects that are in the field of view of the cameras can be determined. This allows for the manipulation of these objects using the robot.

Two different calibration schemes have been implemented. The first is based on the work of Yakimovsky and Cunningham [1], at JPL. The second is based on the work of Roger Tsai [2], [3], at IBM. In this report the implementation details of the technique by Roger Tsai are presented.

Contents

1	Introduction	4
1.1	What is Camera Calibration?	4
1.2	Why is Camera Calibration important?	4
1.3	What does Camera Calibration involve?	5
2	Existing techniques for Camera Calibration	5
3	Camera Calibration using a Two-Stage Technique	7
3.1	The four steps of transformation from 3-D world coordinates to camera coordinates	10
3.2	Calculating the location of a three dimensional point using two cameras	12
4	Implementation and Results	13
5	Appendix - List of Routines	20
6	Appendix - Source Code	22

1 Introduction

1.1 What is Camera Calibration?

Camera calibration is the problem of determining the elements that govern the relationship of the transformation between the 2-D image that a camera perceives and the 3-D information of the imaged object. Thus, camera calibration is the process of determining the geometrical and optical characteristics of the camera and the 3-D position and orientation of the camera relative to a world coordinate system.

1.2 Why is Camera Calibration important?

Camera calibration is important for the following reasons.

1. Camera calibration allows the determination of the relative rotation and translation between the lens centered coordinate system and an arbitrarily defined world coordinate system. For the robotic assembly application the cameras are calibrated with respect to the robot base coordinate system. It is assumed that the robot base does not move after the camera calibration is performed. Under this assumption the camera calibration has to be performed once.
2. When the camera is properly calibrated, its 2-D image coordinates can be translated into real world locations for the objects in its field of view. Thus the 3-D location of objects that are in the field of view of the cameras can be determined. This allows the robot to manipulate these objects and thus

perform simple assembly tasks.

1.3 What does Camera Calibration involve?

Camera calibration involves the determination of two kinds of parameters, intrinsic and extrinsic.

- Intrinsic. These parameters characterize the inherent properties of the camera and the optical model used. They are
 1. Apparent Image Center in the frame buffer.
 2. Image X and Y Scale Factors.
 3. Effective Focal Length.
 4. Lens Radial Distortion Coefficients.
- Extrinsic. These parameters determine the position and orientation of the lens center with respect to the world coordinate system. They are
 1. Translation Vector.
 2. Rotation Matrix.

2 Existing techniques for Camera Calibration

Various techniques are available to determine the parameters needed for camera calibration. Any camera calibration procedure that is chosen should satisfy some pre-

specified criteria that are dictated by accuracy, efficiency, versatility etc., [3]. The camera calibration technique should meet the following criteria.

1. Autonomy. The calibration procedure should not require operator intervention such as giving initial guesses for certain parameters, or choosing certain system parameters manually.
2. Accuracy. The camera calibration technique should have the potential of meeting the accuracy requirements dictated by the application.
3. Reasonable Efficiency. The complete camera calibration technique should not include high dimension nonlinear search. Thus, the calibration approach should allow potential for high speed implementation.
4. Versatility. The calibration technique should operate uniformly and autonomously for a wide range of accuracy requirements, optical setups and applications.
5. Use only common off-the-shelf cameras and lenses: The calibration procedure should not require the use of special professional cameras.

The existing camera calibration techniques can be divided into several categories.

They are

1. Techniques involving full scale nonlinear optimization.
2. Techniques involving computing perspective transformation matrix first using solution of linear equations.

3. Two Plane method [5].

4. Geometric Technique.

In his report Tsai [3] discusses the strengths and weaknesses of several existing camera calibration techniques from the above categories in the light of the criteria mentioned before.

3 Camera Calibration using a Two-Stage Technique

Almost all the techniques for camera calibration use the simple pin-hole camera model. Figure 1 shows the pin-hole model which uses a lens centered coordinate system with a front projection plane. Tsai [3] includes a model of the lens distortion in the camera pin-hole model from the Manual of Photogrammetry [4]. There are two kinds of distortion, radial and tangential. For each kind of distortion, an infinite series is required. According to Tsai [3] only radial distortion needs to be considered and two terms of the series suffice. Figure 2 shows the geometry of the pin-hole camera model with radial lens distortion. The two stages involve calculating the camera intrinsic and extrinsic parameters based on a number of points whose object coordinates in the (X_w, Y_w, Z_w) coordinate system are known, and whose image coordinates (X, Y) are measured.

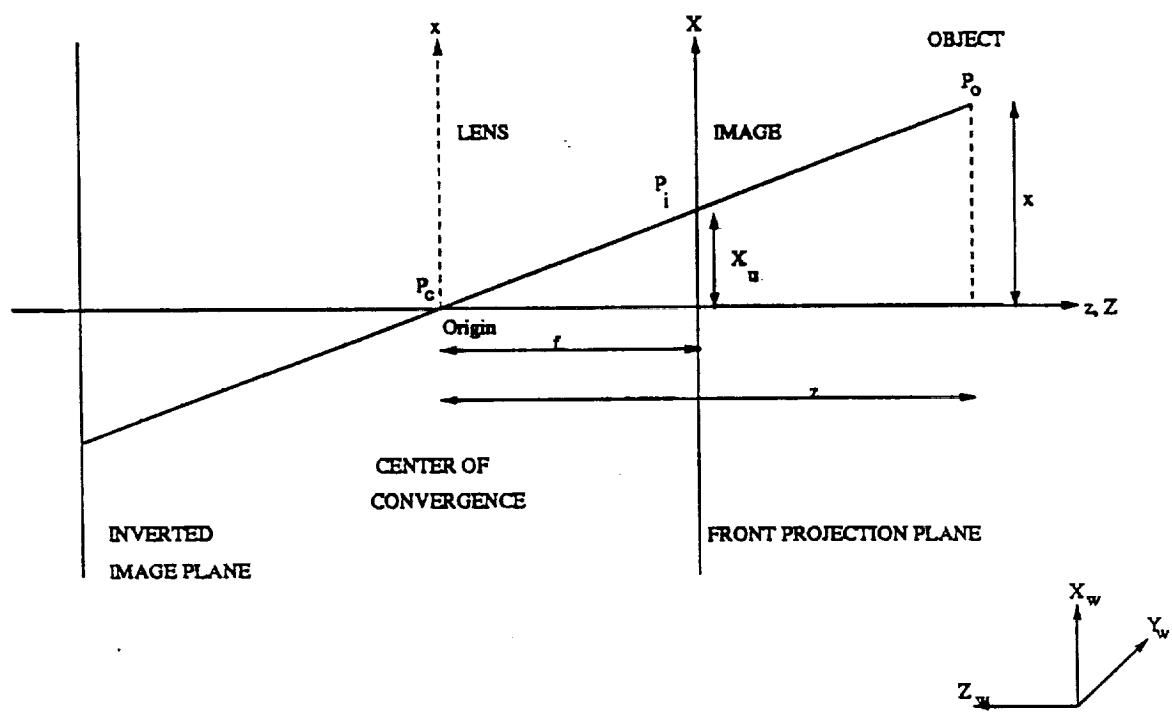


Figure 1: Lens centered coordinate system with front projection.

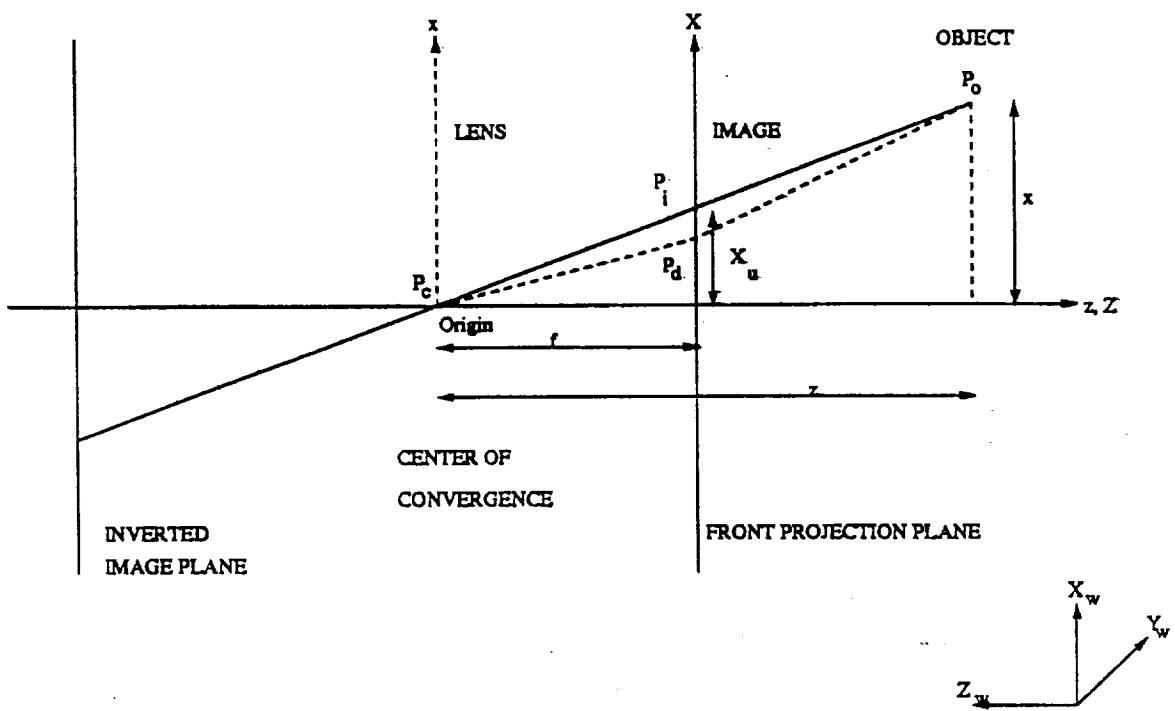


Figure 2: Camera geometry with front projection and radial lens distortion.

3.1 The four steps of transformation from 3-D world coordinates to camera coordinates

In Figure 2 (X_w, Y_w, Z_w) is the 3-D coordinate of the object point P_o in the world coordinate system. (x, y, z) is the 3-D coordinate of the object point P_o in the 3-D camera coordinate system which is centered at the center of the lens, marked Origin in the figure. (X, Y) is the image coordinate system centered at the intersection of the optical axis z and the front image plane. The focal length, f , is the distance between the front image plane and the optical center at the origin. (X_u, Y_u) is the image coordinate of (x, y, z) if a perfect pin-hole camera model is used and (X_d, Y_d) are the coordinates of the image due to radial lens distortion. However, since the units of measurement for (X_f, Y_f) , the coordinates used in the computer, is the number of pixels for the discrete image in the frame memory, additional (conversion) parameters need to be specified and calibrated. These parameters relate the image coordinate in the front image plane to the computer image coordinate system. The four steps are as follows.

Step 1. Rigid body transformation from the object world coordinate system (X_w, Y_w, Z_w) to the camera 3-D coordinate system (x, y, z) .

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} + T \quad (1)$$

where (x, y, z) are the 3-D camera coordinates, (X_w, Y_w, Z_w) are the 3-D world

coordinates, R is a 3×3 rotation matrix

$$R \equiv \begin{bmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{bmatrix} \quad (2)$$

and T is a 3×1 translation vector

$$T \equiv \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \quad (3)$$

Parameters to be calibrated: R and T.

Step 2. Transformation from 3-D camera coordinate (x, y, z) to the ideal (undistorted) image coordinate (X_u, Y_u) using perspective projection with pin hole camera geometry.

$$X_u = f \frac{x}{z} \quad (4)$$

$$Y_u = f \frac{y}{z} \quad (5)$$

Parameter to be calibrated: effective focal length, f.

Step 3. Radial lens distortion.

$$X_d + D_x = X_u \quad (6)$$

$$Y_d + D_y = Y_u \quad (7)$$

where (X_d, Y_d) are the distorted or true image coordinates, (X_u, Y_u) are the undistorted or ideal image coordinates, and D_x and D_y are as given below:

$$D_x = X_d(\kappa_1 r^2 + \kappa_2 r^4) \quad (8)$$

$$D_y = Y_d(\kappa_1 r^2 + \kappa_2 r^4) \quad (9)$$

$$r = \sqrt{X_d^2 + Y_d^2} \quad (10)$$

where κ_1 and κ_2 are the distortion coefficients.

Parameters to be calibrated: distortion coefficients κ_1 and κ_2 .

Step 4. Real image coordinate (X_d, Y_d) to computer image coordinate (X_f, Y_f) transformation.

$$X_f = \frac{s_x X_d}{d_x} \frac{N_{fx}}{N_{cx}} + C_x \quad (11)$$

$$Y_f = \frac{Y_d}{d_y} + C_y \quad (12)$$

where (C_x, C_y) is the computer image coordinate for the origin in the image plane, d_x is the center to center distance between adjacent sensor elements in the X (scan line) direction, d_y is the center to center distance between adjacent sensor elements in the Y direction, N_{cx} is the number of sensor elements in the X direction, N_{fx} is the number of pixels in a line as sampled by the computer, and s_x is the uncertainty image scale factor.

Parameters to be calibrated: uncertainty image scale factor s_x and image coordinate (C_x, C_y) .

3.2 Calculating the location of a three dimensional point using two cameras

Once each camera has been calibrated and there is a way to provide corresponding image coordinates from both the cameras for the same image point (i.e., image cor-

respondence problem is resolved), the location of the 3-D point can be determined.

It can be shown that the world coordinates of the 3-D point being viewed can be calculated using the following equation:

$$\begin{bmatrix} (r_{r_1} X_{u_1} - f_1 r_{11}) & (r_{s_1} X_{u_1} - f_1 r_{21}) & (r_{g_1} X_{u_1} - f_1 r_{31}) \\ (r_{r_1} Y_{u_1} - f_1 r_{41}) & (r_{s_1} Y_{u_1} - f_1 r_{51}) & (r_{g_1} Y_{u_1} - f_1 r_{61}) \\ (r_{r_2} X_{u_2} - f_2 r_{12}) & (r_{s_2} X_{u_2} - f_2 r_{22}) & (r_{g_2} X_{u_2} - f_2 r_{32}) \\ (r_{r_2} Y_{u_2} - f_2 r_{42}) & (r_{s_2} Y_{u_2} - f_2 r_{52}) & (r_{g_2} Y_{u_2} - f_2 r_{62}) \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} = \begin{bmatrix} (f_1 t_{x_1} - t_{r_1} X_{u_1}) \\ (f_1 t_{y_1} - t_{r_1} Y_{u_1}) \\ (f_2 t_{x_2} - t_{r_2} X_{u_2}) \\ (f_2 t_{y_2} - t_{r_2} Y_{u_2}) \end{bmatrix} \quad (13)$$

or, in more compact vector notation,

$$Ax = b \quad (14)$$

Given an overdetermined system of equations as in (14), the technique of Least Mean Square Error can be used to determine an estimate \hat{x} of x . The computation of \hat{x} was performed using a technique known as singular value decomposition [6].

4 Implementation and Results

Figure 3 shows the calibration target used for calibrating the cameras. The target consists of twenty four black circles on a white background. The technique used in this implementation is one of using Monoview Non-coplanar points. The calibration target is moved to three different heights by moving the robot in the world Z-coordinate direction. An algorithm based on eight-connectivity for objects and four-connectivity for holes is used to label the circles and find the centroid of each circle [7]. The

circles on the target are labelled from left to right and from top to bottom as the image is raster scanned. Figure 4 shows the errors in the x, y and z coordinates of the centroids of the circles on the calibration target. The errors are calculated as a difference between the value of the coordinates obtained using the calibration parameters and the value obtained from querying VAL-II regarding the coordinates of the point.

The hardware used for this implementation was the Matrox MVP-VME mounted in a SUN 3/260. This board was used for grabbing the images and for some low level processing. The images were acquired using two overhead Javelin black/white CCD (solid state) cameras (model JE2362). The dimensions of the CCD array are 8.8mm by 6.6mm. The cameras were mounted about 2 meters above the work surface. The calibration parameters obtained for the two cameras are

For Camera 1.

$$R_1 = \begin{bmatrix} 0.960883 & -0.020201 & 0.276217 \\ -0.109655 & -0.928306 & 0.355279 \\ 0.249237 & -0.371670 & -0.894209 \end{bmatrix} \quad (15)$$

$$T_1 = \begin{bmatrix} 264.499399 \\ 896.038140 \\ 1962.189803 \end{bmatrix} \quad (16)$$

$$\kappa_{11} = -0.003782 \quad (17)$$

$$\kappa_{12} = 0.000169 \quad (18)$$

$$f_1 = 41.335538 \quad (19)$$

$$sx_1 = 1.063216 \quad (20)$$

For Camera 2.

$$R_2 = \begin{bmatrix} 0.998254 & -0.058528 & -0.008006 \\ -0.057960 & -0.996370 & 0.062342 \\ -0.011626 & -0.061769 & -0.998023 \end{bmatrix} \quad (21)$$

$$T_2 = \begin{bmatrix} 160.437923 \\ 812.936180 \\ 1091.812649 \end{bmatrix} \quad (22)$$

$$\kappa_{21} = -0.004276 \quad (23)$$

$$\kappa_{22} = 0.000251 \quad (24)$$

$$f_2 = 32.237027 \quad (25)$$

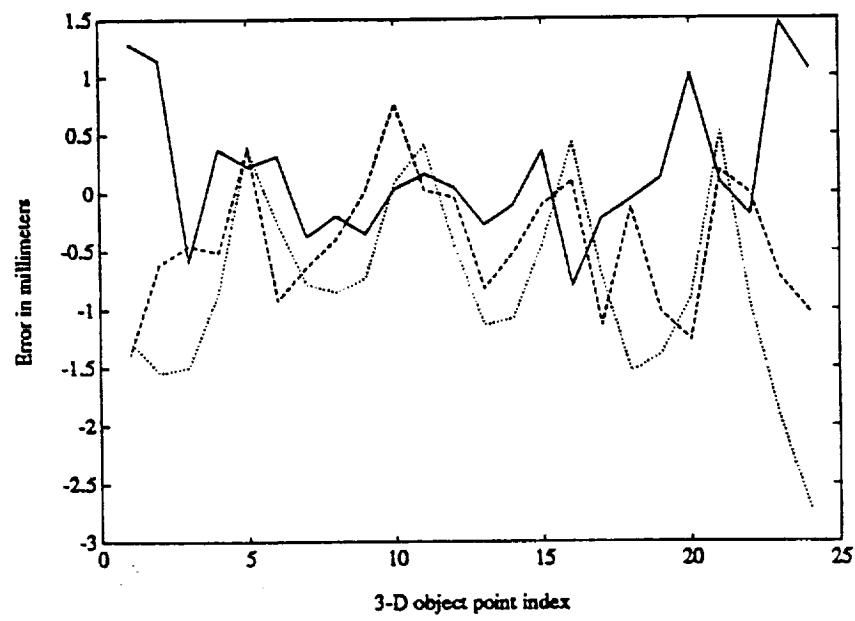


Figure 3: Errors in the x, y, and z coordinates.

$$sx_2 = 1.055226 \quad (26)$$

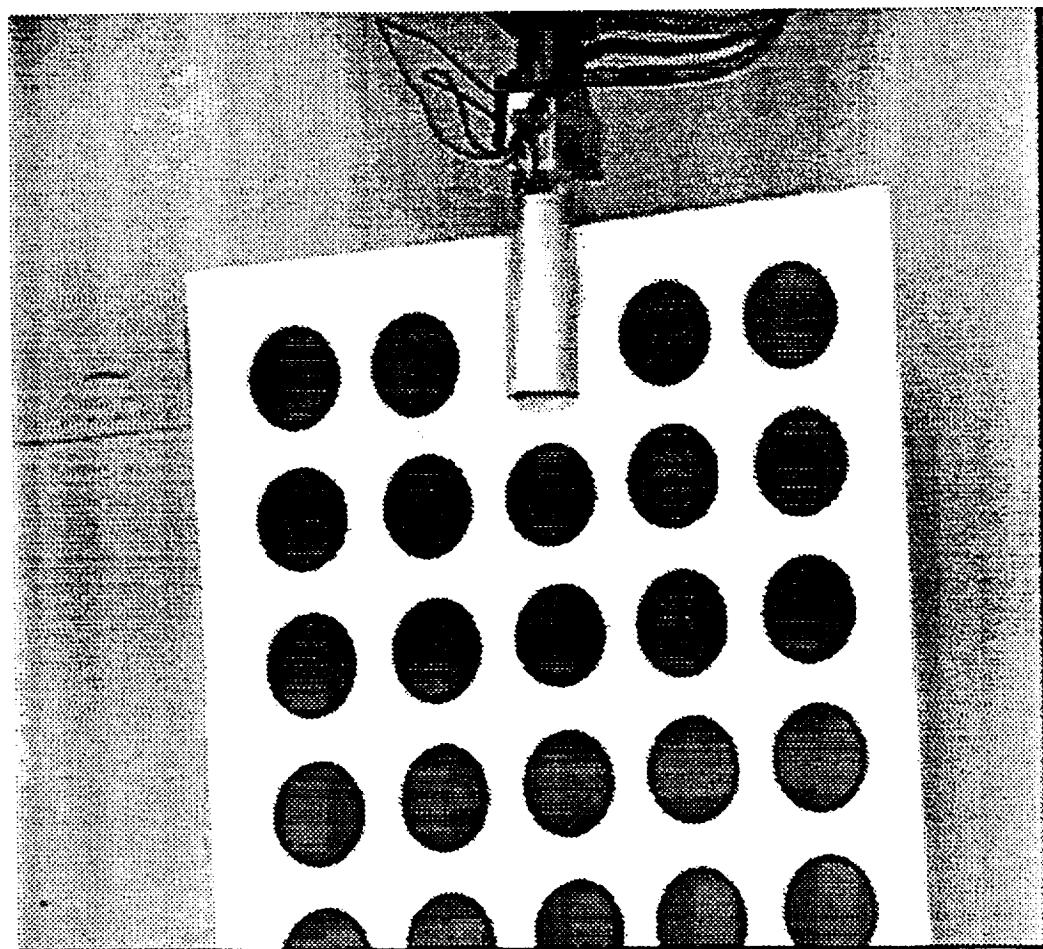


Figure 3: Camera calibration target being held in the gripper.

References

- [1] Y. Yakimovsky and R. Cunningham, "A System for Extracting Three-Dimensional Measurements from a Stereo Pair of TV Cameras," *Computer Graphics and Image Processing*, Vol. 7, pp. 195-210, 1978.
- [2] R. Y. Tsai, "A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 4, pp. 323-344, August 1987.
- [3] R. Y. Tsai, "A Versatile Camera Calibration Technique for High Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses," IBM RC 11413, October 1985.
- [4] "Manual of Photogrammetry," fourth edition, American Society of Photogrammetry, 1980.
- [5] H. Martins, J. Birk, and R. B. Kelley, "Camera Models Based on Data from Two Calibration Planes," *Computer Graphics and Image Processing*, Vol. 17, pp.173-180, 1981.
- [6] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, Numerical Recipes in C, pp. 60-71, 534-536, Cambridge University Press, 1990.

[7] M. C. Repko "An implementation of the blob-labelling algorithm," Internal Communication, 1988.

[8] A. Isaguirre "An implementation of the Supervisor under VAL-II for the VAX 11-750," Private Communication, 1988.

5 Appendix - List of Routines

1. **cal.h:** This header file defines the structures used in the calibration programs and in the blob labeling routine.
2. **ext.h:** This header file contains the definitions of the external variables.
3. **puma.h:** This header file defines the structure used in the programs that interface with the supervisor.
4. **bithresh.c:** This routine uses the Matrox board to threshold images.
5. **calc_3d.c:** This program calculates the 3-D coordinates of a point, given the x and y image coordinates from two different cameras.
6. **calc_params.c:** This routine calculates the camera parameters based on the Monoview Non-Coplanar points technique of Roger Tsai.
7. **calibration.c:** This is the main program which calls all the other routines. It determines the homogeneous transformation matrix linking the camera to the robot base.
8. **cnt_grab.c:** This routine sets up the camera in a continuous grab mode used for positioning the calibration block in the field of view of the cameras.
9. **domatrox.c:** This is the main Matrox routine that calls other routines so that images can be acquired from the cameras.

10. **get_calpts.c**: This routine returns an ordered array of centroids of the calibration points. The centroids are ordered from left to right and top to bottom.
11. **img_grab.c**: This routine is used to grab an image from a camera.
12. **matching.c**: This routine orders the centroids from left to right and top to bottom.
13. **newmomt.c**: This routine is used to label the connected regions in an image and find the centroids of these regions.
14. **read_params.c**: This routine reads in the calibration parameters from a file.
15. **save_params.c**: This routine saves the calibration parameters into a file.
16. **svbksb.c**: This routine is used in conjunction with the singular value decomposition routine.
17. **svdcmp.c**: This routine was translated from its Fortran version in [6].
18. **where.c**: This routine uses the supervisor mode to determine the position of the robot end effector [8].

6 Appendix - Source Code

```
#include <stdio.h>
#include <math.h>
#include "/usr3/mvp/header/im2to3.h"

#define OBJS      24
#define ZTIMES    3
#define IMGX      512
#define IMGY      480
#define PICBYTES  IMGX*IMGY
#define FRAMEBYTES 512*512
#define PI        3.141593
#define TOL       0.000001
#define fncmin(a,b) ((a) <= (b) ? (a) : (b))
#define fncmax(a,b) ((a) >= (b) ? (a) : (b))
#define sign_op(x) (((x) > 0) ? (1) : (-1))

typedef struct dummy {
    double rot[9];
    double trans[3];
    double focal;
    double dist[2];
    double scale;
    double ncx;
    double dx;
    double dy;
    double dxprime;
} CAM_PARAMS;

typedef struct {
    int cx;
    int cy;
    double nfx;
} FRAME_PARAMS;

typedef struct {
    float m00;
    float m10;
    float m01;
    float m11;
    float m20;
    float m02;
    float gravx;
    float gravy;
    float inter;
    float perim;
    float rot;
    char nom[100];
    int xmin;
    int ymin;
    int xmax;
    int ymax;
} MOMENTS;
```

ext.h Sun Nov 19 16:36:44 1989

1

```
extern int *hostimg;
extern int offx[OBJS], offy[OBJS];
extern FRAME_PARAMS *framep;
```

puma.h Sat Nov 18 18:24:29 1989

1

```
struct location {
    char *quality;
    float x;
    float y;
    float z;
    float o;
    float a;
    float t;
};

#define PUMA_SUPER_DEV  "/dev/ttya"
```

```
#include "cal.h"

bithresh(sbuf,dbuf,tval)
int sbuf,dbuf,tval;
{
    short buf[256];
    char lbuf[256];
    int temp;

    temp = tval + 1;
    scaling(0,0,tval,0,buf);
    scaling(temp,255,255,255,buf);
    cwb(buf,lbuf,NULL,256);
    wlut(0,1,0,256,lbuf,lbuf,lbuf);
    slut(0,1);
    inmap(sbuf,dbuf,0);

}
```

```
*****
 * This program calculates the 3D coordinates of a point, given the x and *
 * y image coordinates from two different cameras. The routine also uses *
 * the calibration parameters associated with the cameras. The user must *
 * make sure that the image points from the first camera are passed in the *
 * xandy1 variable and the first camera parameters are passed in the *
 * structure cam1. Camera2 must be handled in a similar manner. It is up *
 * to the user to make sure of point correspondence. Also specify the *
 * number of objects whose 3D coordinates are being calculated using the *
 * variable numobs.
*****
#include "cal.h"
#include "ext.h"

int calc_3d(xandy1,xandy2,numobs,cam1,cam2,coord3d)
int numobs;
double coord3d[][3],xandy1[][2],xandy2[][2];
CAM_PARAMS *cam1,*cam2;
{
double *w, *v,x1,y1,x2,y2,xd,yd,r;
double lineqn[12],b[4],temp[3];
int i,j;

/* Set aside workbuffers for svdcmp and svbksb. */
if( ( w = (double *) malloc( sizeof(double)*200) ) == NULL)
    return(-1);
if( ( v = (double *) malloc( sizeof(double)*200) ) == NULL)
    return(-1);

/* Solve for the 3d world coordinates. */
for (i=0;i<numobs;i++)
{
    xd = (cam1->dxprime/cam1->scale)*(xandy1[i][0] - framep->cx);
    yd = cam1->dy*(xandy1[i][1] - framep->cy);
    r = sqrt(xd*xd + yd*yd);
    x1 = xd + xd*(cam1->dist[0]*r*r + cam1->dist[1]*r*r*r*r);
    y1 = yd + yd*(cam1->dist[0]*r*r + cam1->dist[1]*r*r*r*r);

    xd = (cam2->dxprime/cam2->scale)*(xandy2[i][0] - framep->cx);
    yd = cam2->dy*(xandy2[i][1] - framep->cy);
    r = sqrt(xd*xd + yd*yd);
    x2 = xd + xd*(cam2->dist[0]*r*r + cam2->dist[1]*r*r*r*r);
    y2 = yd + yd*(cam2->dist[0]*r*r + cam2->dist[1]*r*r*r*r);

    lineqn[0] = cam1->focal*cam1->rot[0] - cam1->rot[6]*x1;
    lineqn[1] = cam1->focal*cam1->rot[1] - cam1->rot[7]*x1;
    lineqn[2] = cam1->focal*cam1->rot[2] - cam1->rot[8]*x1;
    lineqn[3] = cam1->focal*cam1->rot[3] - cam1->rot[6]*y1;
    lineqn[4] = cam1->focal*cam1->rot[4] - cam1->rot[7]*y1;
    lineqn[5] = cam1->focal*cam1->rot[5] - cam1->rot[8]*y1;
    lineqn[6] = cam2->focal*cam2->rot[0] - cam2->rot[6]*x2;
    lineqn[7] = cam2->focal*cam2->rot[1] - cam2->rot[7]*x2;
    lineqn[8] = cam2->focal*cam2->rot[2] - cam2->rot[8]*x2;
    lineqn[9] = cam2->focal*cam2->rot[3] - cam2->rot[6]*y2;
    lineqn[10] = cam2->focal*cam2->rot[4] - cam2->rot[7]*y2;
    lineqn[11] = cam2->focal*cam2->rot[5] - cam2->rot[8]*y2;
    b[0] = cam1->trans[2]*x1 - cam1->focal*cam1->trans[0];
    b[1] = cam1->trans[2]*y1 - cam1->focal*cam1->trans[1];
    b[2] = cam2->trans[2]*x2 - cam2->focal*cam2->trans[0];
    b[3] = cam2->trans[2]*y2 - cam2->focal*cam2->trans[1];

    svdcmp(lineqn,4,3,w,v);
```

calc_3d.c

Mon Nov 20 22:43:18 1989

2

```
    svbksb(lineqn,w,v,4,3,b,temp);
    for(j=0;j<3;j++)
        coord3d[i][j] = temp[j];
}
/* Free up memory used by malloc.
   free(w);
   free(v);
   return(0);
}
```

```
*****
 * This routine calculates the camera parameters. The parameters are *
 * determined using the method of Roger Tsai from IBM. The PUMA must hold *
 * the calibration plane. Furthermore, the calibration plane must be held *
 * in at least two different positions. The positions do not necessarily *
 * have to differ in the x and y directions but the z MUST change. *
 * Initially, sx = 1.0 and k1 = 0.0 and k2 = 0.0. In this routine we get *
 * the final value for sx and an initial value for Tz and f. Then we solve*
 * for exact values of Tz, f, k1, k2 by minimizing the sum of squares. *
 * This routine then places final values in the struct camp. *
 ****
#include "cal.h"
#include "ext.h"

int calc_params(camera,camp)
int camera;
CAM_PARAMS *camp;
{
int height,m,n,i,cnt,index,index2,iter,tmp,status;
float base[6];
double xandy[OBJS][2],x1,y1,xdtmp,ydtmp;
double xd[ZTIMES*OBJS],yd[ZTIMES*OBJS];
double x3,y3,z3,zd,rd,rd3,rd5,rc,final,final_prev,err;
double xg[ZTIMES*OBJS],yg[ZTIMES*OBJS];
double xw[ZTIMES*OBJS],yw[ZTIMES*OBJS],zw[ZTIMES];
double uk[2],uf[2],b[ZTIMES*OBJS*2],w2[ZTIMES*OBJS],y2[ZTIMES*OBJS];
double lineqn[ZTIMES*OBJS*7],a[7];
double *w, *v;
double junk1,junk2;

/* Set aside space for work buffers w and v in svdcmp and svbksb. */
if( ( w = (double *) malloc( sizeof(double)*200) ) == NULL)
{
    printf("Assigned null pointer to variable w.\n");
    return(-1);
}

if( ( v = (double *) malloc( sizeof(double)*200) ) == NULL)
{
    printf("Assigned null pointer to variable v.\n");
    return(-1);
}

m = OBJS*ZTIMES;
n = 7;

/* Several changes in position of the calibration block are necessary. */
/* Therefore the PUMA is moved ZTIMES. A minimum of two changes */
/* are necessary and more changes will give hopefully more accurate info.*/
for(height=0;height<ZTIMES;height++)
{
    if((status = get_calpts(height*PICTBYTES,camera,xandy,base)) == -1)
    {
        printf("Error in acquiring calibration points.\n");
        return(-1);
    }

    /* Take out the ordered centroids and put in the arrays - xd and yd. */
    zw[height] = (double) base[2];
    for (i=0;i<OBJS;i++)
    {
        index = i+height*OBJS;
        index2 = i*n + height*OBJS*n;
    }
}
```

```
xg[index] = (xandy[i][0] - framep->cx);
yg[index] = (xandy[i][1] - framep->cy);
xd[index] = (1.0/camp->scale)*(camp->dxprime)*xg[index];
yd[index] = camp->dy*yg[index];
xw[index] = (double) (base[0]+offx[i]);
yw[index] = (double) (base[1]+offy[i]);
lineqn[index2]=yd[index]*xw[index];
lineqn[index2+1]=yd[index]*yw[index];
lineqn[index2+2]=yd[index]*zw[height];
lineqn[index2+3]=yd[index];
lineqn[index2+4]=-xd[index]*xw[index];
lineqn[index2+5]=-xd[index]*yw[index];
lineqn[index2+6]=-xd[index]*zw[height];
}
}
svdcmp(lineqn,m,n,w,v);
svbksb(lineqn,w,v,m,n,xd,a);
for (i=0;i<7;i++)
    printf("The a values are %4.8lf\n",a[i]);

/* Determine the magnitude of Ty and the value of sx. */
camp->trans[1] = 1.0/(sqrt (a[4]*a[4] + a[5]*a[5] + a[6]*a[6]));
camp->scale = (sqrt (a[0]*a[0] + a[1]*a[1] + a[2]*a[2]))*camp->trans[1];

/* Determine the sign of Ty. To start with, assume the sign of Ty is +1.*/
/* Also determine Tx,r0,r1,r3,r4,r5 */
camp->rot[0]=a[0]*(camp->trans[1]/camp->scale);
camp->rot[1]=a[1]*(camp->trans[1]/camp->scale);
camp->rot[2]=a[2]*(camp->trans[1]/camp->scale);
camp->rot[3]=a[4]*camp->trans[1];
camp->rot[4]=a[5]*camp->trans[1];
camp->rot[5]=a[6]*camp->trans[1];
camp->trans[0] = a[3]*(camp->trans[1]/camp->scale);

/* test if r2 = sqrt(1 - r0*ro - r1*rl) */
junk1 = sqrt(1.0 - camp->rot[0]*camp->rot[0] - camp->rot[1]*camp->rot[1]);
junk2 = sqrt(1.0 - camp->rot[3]*camp->rot[3] - camp->rot[4]*camp->rot[4]);
printf("r2,r5 as a result of svdcmp %6.15lf %6.15lf\n",camp->rot[2],camp->rot[5]);
printf("r2,r5 as a result of calcs. %6.15lf %6.15lf\n",junk1,junk2);
printf("delta r2,r5 %6.15lf %6.15lf\n", (junk1-camp->rot[2]), (junk2-camp->rot[5]));

tmp = OBJS - 1;
x1=camp->rot[0]*xw[tmp]+camp->rot[1]*yw[tmp]
    +camp->rot[2]*zw[0]+camp->trans[0];
y1=camp->rot[3]*xw[tmp]+camp->rot[4]*yw[tmp]
    +camp->rot[5]*zw[0]+camp->trans[1];

if(! ((sign_op(x1) == sign_op(xg[tmp]))&&(sign_op(y1) == sign_op(yg[tmp]))))
{
    camp->trans[0] = -camp->trans[0];
    camp->trans[1] = -camp->trans[1];
    for(cnt=0;cnt<6;cnt++)
        camp->rot[cnt] = -camp->rot[cnt];
}

/* We have determined r0,r1,r3,r4,r5,r6 of the 9x9 rotation matrix. */
/* Now we can determine the remaining values from the cross product. */
camp->rot[6]=camp->rot[1]*camp->rot[5]-camp->rot[2]*camp->rot[4];
camp->rot[7]=camp->rot[2]*camp->rot[3]-camp->rot[0]*camp->rot[5];
camp->rot[8]=camp->rot[0]*camp->rot[4]-camp->rot[1]*camp->rot[3];

/* The last step is to determine Tz and the focal length. To solve */
/* for these parameters, the calibration plane must not be exactly */
/* parallel to the image plane. */
```

```
n = 2;
for (height=0; height<ZTIMES; height++)
{
    for (i=0; i<OBJS; i++)
    {
        index = i+height*OBJS;
        index2 = i*n + height*OBJS*n;
        y2[index] = camp->rot[3]*xw[index]+camp->rot[4]*yw[index]
                    +camp->rot[5]*zw[height]+camp->trans[1];
        w2[index] = camp->rot[6]*xw[index]+camp->rot[7]*yw[index]
                    +camp->rot[8]*zw[height];
        lineqn[index2] = y2[index];
        lineqn[index2+1] = -1.0*camp->dy*yg[index];
        b[index] = w2[index]*camp->dy*yg[index];
    }
}
svdcmp(lineqn, m, n, w, v);
svbkbsb(lineqn, w, v, m, n, b, uf);

/* We have determined an initial estimate for Tz and f. Now solving for */
/* k1 and k2 we can substitute back and forth and converge on an exact */
/* solution for k1, k2, f, and Tz. */
err = 10.0;
final_prev = 0.0;
iter = 0;

printf("We are going to solve for exact solutions for Tz, f, k1, and k2.\n");
printf("This will take many iterations and some time. Go get some coffee.\n");
while(fabs(err) > TOL)
{
    iter++;
    printf("Iteration = %d\r", iter);
    fflush(stdout);

/* Determine values for k1 and k2. */
m= OBJS*ZTIMES;
n=2;
for (height=0; height<ZTIMES; height++)
{
    for (i=0; i<OBJS; i++)
    {
        index = i+height*OBJS;
        index2 = i*n + height*OBJS*n;
        xdtmp = (1.0/camp->scale)*(camp->dxprime)*xg[index];
        ydtmp = camp->dy*yg[index];
        x3=camp->rot[0]*xw[index]+camp->rot[1]*yw[index]
            +camp->rot[2]*zw[height]+camp->trans[0];
        y3=camp->rot[3]*xw[index]+camp->rot[4]*yw[index]
            +camp->rot[5]*zw[height]+camp->trans[1];
        zd=camp->rot[6]*xw[index]+camp->rot[7]*yw[index]+camp->rot[8]*zw[height];
        z3= zd+uf[1];
        rc = sqrt(x3*x3 + y3*y3);
        rd = sqrt(xdtmp*xdtmp + ydtmp*ydtmp);
        rd3 = rd*rd*rd;
        rd5 = rd3*rd*rd;

        lineqn[index2] = rd3*z3;
        lineqn[index2+1] = rd5*z3;
        b[index] = (-rd*z3 + uf[0]*rc);
    }
}
svdcmp(lineqn, m, n, w, v);
svbkbsb(lineqn, w, v, m, n, b, uk);
```

```
/* Determine values for f and Tz. */  
m= OBJS*ZTIMES;  
n=2;  
for (height=0; height<ZTIMES; height++)  
{  
    for (i=0; i<OBJS; i++)  
    {  
        index = i+height*OBJS;  
        index2 = i*n + height*OBJS*n;  
        xdtmp = (1.0/camp->scale)*(camp->dxprime)*xg[index];  
        ydtmp = camp->dy*yg[index];  
        x3=camp->rot[0]*xw[index]+camp->rot[1]*yw[index]  
            +camp->rot[2]*zw[height]+camp->trans[0];  
        y3=camp->rot[3]*xw[index]+camp->rot[4]*yw[index]  
            +camp->rot[5]*zw[height]+camp->trans[1];  
        zd=camp->rot[6]*xw[index]+camp->rot[7]*yw[index]+camp->rot[8]*zw[height];  
        rc = sqrt(x3*x3 + y3*y3);  
        rd = sqrt(xdtmp*xdtmp + ydtmp*ydtmp);  
        rd3 = rd*rd*rd;  
        rd5 = rd3*rd*rd;  
        lineqn[index2]= (-rc);  
        lineqn[index2+1]= (rd + rd3*uk[0] + rd5*uk[1]);  
        b[index]= (-lineqn[index2+1]*zd);  
    }  
}  
svdcmp(lineqn,m,n,w,v);  
svbksb(lineqn,w,v,m,n,b,uf);  
  
/* Determine value of function we are minimizing in order to get exact values */  
/* for k1, k2, f, and Tz. The difference between two consecutive values is */  
/* used to stop the iterations. The variable TOL is used for this purpose. */  
final = 0.0;  
for (height=0; height<ZTIMES; height++)  
{  
    for (i=0; i<OBJS; i++)  
    {  
        index = i+height*OBJS;  
        xdtmp = (1.0/camp->scale)*(camp->dxprime)*xg[index];  
        ydtmp = camp->dy*yg[index];  
        x3=camp->rot[0]*xw[index]+camp->rot[1]*yw[index]  
            +camp->rot[2]*zw[height]+camp->trans[0];  
        y3=camp->rot[3]*xw[index]+camp->rot[4]*yw[index]  
            +camp->rot[5]*zw[height]+camp->trans[1];  
        zd=camp->rot[6]*xw[index]+camp->rot[7]*yw[index]+camp->rot[8]*zw[height];  
        z3= zd+uf[1];  
        rc = sqrt(x3*x3 + y3*y3);  
        rd = sqrt(xdtmp*xdtmp + ydtmp*ydtmp);  
        final = (rd*(1 + uk[0]*rd*rd + uk[1]*rd*rd*rd*rd)*z3 - uf[0]*rc)*  
            (rd*(1 + uk[0]*rd*rd + uk[1]*rd*rd*rd*rd)*z3 - uf[0]*rc) +  
            final;  
    }  
}  
err = final - final_prev;  
final_prev = final;  
} /*end while loop*/  
  
camp->trans[2] = uf[1];  
camp->focal = uf[0];  
camp->dist[0] = uk[0];  
camp->dist[1] = uk[1];  
free(w);  
free(v);  
return(0);
```

calc_params.c

Tue Dec 5 17:57:05 1989

5

}

```
*****  
/* Program Calibration */  
/*  
 * The purpose of this program is the calibration of a camera using  
 * a monoview NONcoplanar set of points. The program will determine the  
 * homogeneous transformation matrix linking the camera to the robot  
 * base. It will also determine the focal length of the lens. This  
 * program is based on the following article:  
 * Roger Tsai, "A Versatile Camera Calibration Technique for High  
 * Accuracy 3D Machine Vision Metrology Using Off-the-Shelf Cameras and  
 * Lenses," IEEE Journal of Robotics and Automation, Vol. RA-3, No. 4,  
 * August 1987.  
 * NOTE: All dimensions must be in millimeters.  
 * This calibration procedure uses a calibration plate with  
 * 24 calibration dots (40mm in diameter) arranged on a  
 * square grid (55mm -in x and y directions- between grid points).  
 */  
/* AUTHORS: Michael C. Repko and Deepak Sood */  
/* DATE : Nov. 1989 */  
*****  
#include "cal.h"  
  
int *hostimg;  
int offy[] =  
{220,220,220,220,165,165,165,165,110,110,110,110,110,  
 55,55,55,55,0,0,0,0};  
/* {0,0,0,0,55,55,55,55,110,110,110,110,110,165,165,165,165,  
 220,220,220,220}; */  
int offx[] =  
{-110,-55,0,55,110,-110,-55,0,55,110,-110,-55,0,55,110,-110,-55,  
 0,55,110,-110,-55,55,110};  
/* {110,55,-55,-110,110,55,0,-55,-110,110,55,0,-55,-110,110,55,  
 0,-55,-110,110,55,0,-55,-110}; */  
FRAME_PARAMS *framep;  
  
main()  
{  
int i,status;  
float world[6];  
char resp;  
CAM_PARAMS *campl, *camp2;  
double coords[OBJS][3],xy1[OBJS][2],xy2[OBJS][2];  
double tmp1,tmp2,tmp3;  
FILE *out;  
  
/* Set aside space for the pointers to CAM_PARAMS and FRAME_PARAMS. */  
campl = (CAM_PARAMS *) malloc(sizeof(CAM_PARAMS));  
camp2 = (CAM_PARAMS *) malloc(sizeof(CAM_PARAMS));  
framep = (FRAME_PARAMS *) malloc(sizeof(FRAME_PARAMS));  
  
/* Initialize the FRAME_PARAMS to the correct values for the Matrox */  
/* frame grabber. */  
framep->cx = 255;  
framep->cy = 255;  
framep->nfx = 512.0;  
  
/* Initialize the appropriate CAM_PARAMS to their correct values, */  
/* using Javelin camera information. Note, not all elements are */  
/* initialized - this won't cause a problem later on. */  
/* First camera parameters. */  
campl->ncx = 576.0;  
campl->dx = 8.8/576.0;  
campl->dy = 6.6/485.0;  
campl->dxprime = campl->dx * (campl->ncx/framep->nfx);
```

```
    camp1->scale = 1.0;
/* Second camera parameters.                                         */
camp2->ncx = 576.0;
camp2->dx = 8.8/576.0;
camp2->dy = 6.6/485.0;
camp2->dxprime = camp2->dx * (camp2->ncx/framep->nfx);
camp2->scale = 1.0;

/* Set aside space for hostimg - 2*512*480*sizeof(int).           */
/* hostimg is a global variable and is declared in cal.h.          */
hostimg = (int *) malloc( sizeof(int)*(ZTIMES*PICBYTES));          */

/* Check if calibration has been done already.                      */
printf("Have the parameters already been determined? (y/n)\n");
scanf(" %c",&resp);
getchar();
if ((resp == 'y')||(resp == 'Y'))
{

/* If camera parameters have been determined then read parameters from */
/* the desired file.                                                 */
read_params(camp1,camp2);                                         */

/* Display parameters for first camera.                            */
printf("The parameters for the first camera:\n");
printf("tx,ty,tz = %4.4lf %4.4lf %4.4lf \n",
       camp1->trans[0],camp1->trans[1],camp1->trans[2]);
printf("r0,r1,r2 = %4.4lf %4.4lf %4.4lf \n",
       camp1->rot[0],camp1->rot[1],camp1->rot[2]);
printf("r3,r4,r5 = %4.4lf %4.4lf %4.4lf \n",
       camp1->rot[3],camp1->rot[4],camp1->rot[5]);
printf("r6,r7,r8 = %4.4lf %4.4lf %4.4lf \n",
       camp1->rot[6],camp1->rot[7],camp1->rot[8]);
printf("k1,k2 = %4.6lf %4.6lf\n",
       camp1->dist[0],camp1->dist[1]);
printf("f = %4.4lf \n",camp1->focal);
printf("sx = %4.4lf \n",camp1->scale);

/* Display parameters for second camera.                           */
printf("The parameters for the second camera:\n");
printf("tx,ty,tz = %4.4lf %4.4lf %4.4lf \n",
       camp2->trans[0],camp2->trans[1],camp2->trans[2]);
printf("r0,r1,r2 = %4.4lf %4.4lf %4.4lf \n",
       camp2->rot[0],camp2->rot[1],camp2->rot[2]);
printf("r3,r4,r5 = %4.4lf %4.4lf %4.4lf \n",
       camp2->rot[3],camp2->rot[4],camp2->rot[5]);
printf("r6,r7,r8 = %4.4lf %4.4lf %4.4lf \n",
       camp2->rot[6],camp2->rot[7],camp2->rot[8]);
printf("k1,k2 = %4.6lf %4.6lf\n",
       camp2->dist[0],camp2->dist[1]);
printf("f = %4.4lf \n",camp2->focal);
printf("sx = %4.4lf \n",camp2->scale);

/* Test these parameters with the PUMA and calibration block to      */
/* insure they are correct.                                         */
printf("Testing the parameters with the PUMA and calibration block.\n");
if( (status = get_calpts(0,1,xyl,world)) == -1)
{
    printf("Error acquiring calpts with camera1.\n");
    return;
}
if( (status = get_calpts(0,2,xy2,world)) == -1)
{
    printf("Error acquiring calpts with camera2.\n");
}
```

```

calibration.c      Sun Dec  2 15:29:39 1990      3

    return;
}
if( (status = calc_3d(xy1,xy2,OBJS,camp1,camp2,coords)) == -1)
{
    printf("Error in calculating 3D points.\n");
    return;
}
out = fopen("error.dat","w");
fprintf(out,"xw-calc yw-calc zw-calc xw-real yw-real zw-real\n");
for(i=0;i<OBJS;i++)
{
    tmp1 = (double) (world[0] + offx[i]);
    tmp2 = (double) (world[1] + offy[i]);
    tmp3 = (double) world[2];
    fprintf(out, " %4.4lf %4.4lf %4.4lf %4.4lf %4.4lf\n",
            coords[i][0],coords[i][1],coords[i][2],tmp1,tmp2,tmp3);
}
fclose(out);
}

else
{
/* Calculate the camera parameters, this procedure must be repeated      */
/* for each camera.                                                       */
printf("Calibration of Camera 1.\n");
printf("You must move the PUMA to %d different heights at the prompt.\n",
       ZTIMES);
calc_params(1,camp1);
printf("Calibration of Camera 2.\n");
printf("You must move the PUMA to %d different heights at the prompt.\n",
       ZTIMES);
calc_params(2,camp2);

/* Save the parameters for future use.                                     */
printf("Do you wish to save the parameters?\n");
scanf(" %c", &resp);
getchar();
if ((resp == 'y')||(resp == 'Y'))
{
    save_params(camp1,camp2);
}

/* Display parameters for first camera.                                     */
printf("The parameters for the first camera:\n");
printf("tx,ty,tz = %4.4lf %4.4lf %4.4lf \n",
       camp1->trans[0],camp1->trans[1],camp1->trans[2]);
printf("r0,r1,r2 = %4.4lf %4.4lf %4.4lf \n",
       camp1->rot[0],camp1->rot[1],camp1->rot[2]);
printf("r3,r4,r5 = %4.4lf %4.4lf %4.4lf \n",
       camp1->rot[3],camp1->rot[4],camp1->rot[5]);
printf("r6,r7,r8 = %4.4lf %4.4lf %4.4lf \n",
       camp1->rot[6],camp1->rot[7],camp1->rot[8]);
printf("k1,k2 = %4.6lf %4.6lf\n",
       camp1->dist[0],camp1->dist[1]);
printf("f = %4.4lf \n",camp1->focal);

/* Display parameters for second camera.                                    */
printf("The parameters for the second camera:\n");
printf("tx,ty,tz = %4.4lf %4.4lf %4.4lf \n",
       camp2->trans[0],camp2->trans[1],camp2->trans[2]);
printf("r0,r1,r2 = %4.4lf %4.4lf %4.4lf \n",
       camp2->rot[0],camp2->rot[1],camp2->rot[2]);
printf("r3,r4,r5 = %4.4lf %4.4lf %4.4lf \n",
       camp2->rot[3],camp2->rot[4],camp2->rot[5]);

```

```

calibration.c      Sun Dec  2 15:29:39 1990      4

    printf("r6,r7,r8 = %4.4lf %4.4lf %4.4lf \n",
           camp2->rot[6],camp2->rot[7],camp2->rot[8]);
    printf("k1,k2 = %4.6lf %4.6lf\n",
           camp2->dist[0],camp2->dist[1]);
    printf("f = %4.4lf \n",camp2->focal);

/* Test these parameters with the PUMA and calibration block to      */
/* insure they are correct.                                         */
printf("Testing the parameters with the PUMA and calibration block.\n");
if( (status = get_calpts(0,1,xy1,world)) == -1)
{
    printf("Error acquiring calpts with camera1.\n");
    return;
}
if( (status = get_calpts(0,2,xy2,world)) == -1)
{
    printf("Error acquiring calpts with camera2.\n");
    return;
}
if( (status = calc_3d(xy1,xy2,OBJS,camp1,camp2,coords)) == -1)
{
    printf("Error in calculating 3D points.\n");
    return;
}
out = fopen("error.dat","w");
fprintf(out,"xw-calc yw-calc zw-calc xw-real yw-real zw-real\n");
for(i=0;i<OBJS;i++)
{
    tmp1 = (double) (world[0] + offx[i]);
    tmp2 = (double) (world[1] + offy[i]);
    tmp3 = (double) world[2];
    fprintf(out,"%4.4lf %4.4lf %4.4lf %4.4lf %4.4lf\n",
            coords[i][0],coords[i][1],coords[i][2],tmp1,tmp2,tmp3);
}
fclose(out);
}

/* Free up memory set aside for variables.                                */
free(hostimg);
free(camp1);
free(camp2);
free(framep);
}

```

```
#include "cal.h"
```

```
cnt_grab(cam_a,buf_a)
int cam_a,buf_a;
{
    mvp_install("/dev/mvp0",0);
    init();
    creset();

    opmode(2,buf_a);

    inmode(1);
    outmode(0);

    if(cam_a == 0)
        sync(1,1);
    else
        sync(1,0);
    gain(128);
    offset(50);
    outpath(buf_a,-1,0,0);
    video(1,1);

    opmode(3,buf_a);
    chan(cam_a);
    cgrab(-1);

    mvp_uninstall();
}
```

domatrox.c Mon Apr 30 17:31:29 1990 1

```
#include "cal.h"
#include "ext.h"

domatrox(img_off,camera)
int img_off,camera;
{
/* Variable declaration for Matrox */
    unsigned char matroximg[2*PICBYTES];
    int wbuf[258],done,cnt,thresh,temp;
    char ret;
    unsigned long offset;
    int i,j;

/* Initialize the Matrox board */
    mvp_install("/dev/mvp0",0);
    init();
    creset();

/* Disable video display and clear buffers */
    video(0,1);
    clear(4,0);
    clear(5,0);

/* Set X-axis labelling for histogram to decimal */
    histmode(1);

    printf("This section is for generating a binary image of the \n");
    printf("calibration block.\n");
/* Store image in frame grabber and display image. */
    for(cnt=0;cnt<259;cnt++)
        wbuf[cnt]=0;
    img_grab(camera,0);
    opmode(2,0);
    outpath(0,-1,0,0);
    video(1,1);
    histo(0,wbuf);
    printf("Hit any key and return to continue.\n");
    scanf(" %d",&temp);
    ret = getchar();

/* Display histogram for image. */
    outpath(2,-1,0,0);
    opmode(1,2);
    move(100,300);
    drawhist(128,128,200,0,wbuf,0);
    printf("Hit any key and return to continue.\n");
    scanf(" %d",&temp);
    ret = getchar();

/* Decide on threshold value and display thresholded image. */
    done=0;
    while(!done)
    {
        opmode(2,0);
        printf("What is the threshold value?\n");
        scanf(" %d",&thresh);
        bithresh(0,2,thresh);
        outpath(2,-1,0,0);
        printf("Done with thresholding? (1=yes, 0=no)\n");
        scanf(" %d",&done);
    }

/* Transfer the thresholded image (fbuf = 2) to hostimg in I/O mode. */
}
```

```
    offset = 2*FRAMEBYTES;
    opmode(0,2);
    rd_vram(offset,(unsigned long) PICBYTES, matroximg);
    for (j=0; j<IMGY; j++)
    {
        for (i=0; i<IMGX; i++)
        {
            *(hostimg + i + j*IMGX+img_off)=(0 | ((int) matroximg[i + j*IMGX]));
        }
    }

    mvp_uninstall();

}
```

```

get_calpts.c      Tue Nov 21 14:04:37 1989      1

/*
 * This routine returns an ordered array of the centroids of the calibration *
 * points. The centroids are are ordered from left to right and top to      *
 * bottom.                                                               *
 */
#include "cal.h"
#include "ext.h"

int get_calpts(offset,cam,ordered,base)
int offset,cam;
double ordered[][][2];
float base[];
{
int i;
char resp;
short frame[4],numobs;
MOMENTS *calpts;

/* Set aside memory for the structure calpts. */
if( (calpts = (MOMENTS *) malloc(sizeof(MOMENTS)*200)) == NULL)
{
    printf("Null pointer assigned to calpts.\n");
    return(-1);
}

/* This section allows the user to place the PUMA in a different position. */
/*The monitor is in continuous mode so that the user                      */
/* can make sure the calibration plate is completely in the camera's view. */
printf("If necessary move the PUMA.\n");
printf("The monitor is in continuous mode during positioning.\n");
printf("Hit any key and return when you are done.\n");
cnt_grab(cam,0);
scanf(" %c",&resp);
getchar();
mvp_install("/dev/mvp0",0);
opmode(3,0);
cgrab(0);
mvp_uninstall();
printf("\n");

/* This section finds the position of the robot wrt the robots base.        */
/* Each calibration point is a simple offset from the x,y,z position       */
/* vector which is returned.                                                 */
where(base);
printf("The PUMA x,y,z coords at this position are:\n");
printf("x = %4.2f, y = %4.2f, z = %4.2f.\n",base[0],base[1],base[2]);

/* Call the Matrox routines to grab the images,threshold them, and place      */
/* the resulting image in the array hostimg. An offset into hostimg must      */
/* be specified.                                                       */
domatrox(offset,cam);

/* Set up framesize; Exclude the edges. THIS IS FOR MOMENTS ONLY.           */
/* This is to take into account loss of information due to horizontal and   */
/* vertical retrace.                                                 */
frame[0]=5;
frame[1]=475;
frame[2]=5;
frame[3]=505;

/* Call blob labelling and moment generating routine.                         */
momt((hostimg+offset),frame,calpts,&numobs);
if (numobs != OBJS)
{

```

get_calpts.c Tue Nov 21 14:04:37 1989 2

```
    printf("Error: numobs does not equal OBJS.\n");
    printf("The numobs = %d and OBJS = %d\n",numobs,OBJS);
    return(-1);
}

/* Load centroids into a temporary array.
/* Order the centroids from top to bottom and left to right. */
for (i=0;i<OBJS;i++)
{
    ordered[i][0] = (double) calpts[i+101].gravx;
    ordered[i][1] = (double) calpts[i+101].gravy;
}
matching(ordered);
/* Free up memory used by calpts. */
free(calpts);

return(0);
}
```

img_grab.c Sat Nov 18 18:21:48 1989 1

```
#include "cal.h"

img_grab(cam_a, buf_a)
int cam_a,buf_a;
{
int cnt,dummy++;

/* Put in processing mode. */
    opmode(2,buf_a);

/* Clear the buffer.      */
    clear(buf_a,0);

/* Tell matrox that we are using a B/W camera with 8-bit */
/* grey scaling.                                         */
    inmode(1);
    outmode(0);
/* Set offset and gain.        */
    offset(50);
    gain(128);
/* Select channel/camera input.   */
    chan(cam_a);

/* This mvp_sync() is necessary!    */
    mvp_sync();

/* Select sync for camera. Sync(1,1) external, Sync(1,0) no */
/* external sync.                                         */
    if(cam_a == 0)
        sync(1,1);
    else
        sync(1,0);

/* Wait loop to allow PLL to settle down.                  */
    dummy=0;
    for(cnt=0;cnt<100000;cnt++)
        dummy=dummy+1;
/* Take picture and put in Matrox memory.                 */
    snapshot(buf_a);

/* This mvp_sync() is necessary!    */
    mvp_sync();

}
```

```
- matching.c      Sat Nov 18 18:21:48 1989      1
-
- #include "cal.h"
-
- matching(image1)
- double image1[OBJS][2];
-
- {
-     double cnr1[4][2];
-     double leftline[10][2], rightline[10][2], ordered[100][2];
-     double dist1, distlft;
-     int temp, offset, ext, newcnt, cnt;
-     static int cnr[4][2] =
-     {
-         {0 , 0 },
-         {512, 0 },
-         {512, 512},
-         {0 , 512}
-     };
-
-     /* Find the centroids closest to the corners. */
-     for (ext=0;ext<=3;ext++)
-     {
-         distlft = (512.0*512.0 + 480.0*480.0);
-         for (cnt=0;cnt<OBJS;cnt++)
-         {
-             dist1= (image1[cnt][0]-cnr[ext][0])*(image1[cnt][0]-cnr[ext][0]) +
-                   (image1[cnt][1]-cnr[ext][1])*(image1[cnt][1]-cnr[ext][1]);
-             if(dist1 < distlft)
-             {
-                 distlft = dist1;
-                 cnr1[ext][0] = image1[cnt][0];
-                 cnr1[ext][1] = image1[cnt][1];
-             }
-         }
-     }
-
-     /* Leftmost vertical line of centroids. */
-     new_line(cnr1[0][1],cnr1[0][0],cnr1[3][1],cnr1[3][0],1,
-              &newcnt,leftline,image1);
-
-     /* Rightmost vertical line of centroids. */
-     new_line(cnr1[1][1],cnr1[1][0],cnr1[2][1],cnr1[2][0],1,
-              &newcnt,rightline,image1);
-
-     /* Ordered centroids (left to right - top to bottom). */
-     temp = 0;
-     offset = 0;
-     for(cnt=0;cnt<newcnt;cnt++)
-     {
-         new_line(leftline[cnt][0],leftline[cnt][1],
-                  rightline[cnt][0],rightline[cnt][1],0,
-                  &offset,&ordered[temp][0],image1);
-         temp = offset+temp;
-     }
-
-     /* Return the ordered array. */
-     for(cnt=0;cnt<OBJS;cnt++)
-     {
-         image1[cnt][0] = ordered[cnt][0];
-         image1[cnt][1] = ordered[cnt][1];
-     }
-
- } /* end matching */
```

```
/* Routine that finds the equation of a line and fits points. */

new_line(endlx,endly,end2x,end2y,dir,linecnt,linecoord,imgcoord)
double endlx,endly,end2x,end2y,linecoord[10][2];
double imgcoord[24][2];
int *linecnt,dir;
{
    double slope,inter,ytemp,tempx,tempy;
    int i,j,cnt,odir;

    if(dir)
        odir = 0;
    else
        odir = 1;
    slope = (end2y-endly)/(end2x-endlx);
    inter = endly - endlx*slope;
    *linecnt = 0;
    for(cnt=0;cnt<OBJS;cnt++)
    {
        ytemp = imgcoord[cnt][dir] * slope + inter;
        if(fabs(imgcoord[cnt][odir] - ytemp) < 10.0)
        {
            linecoord[*linecnt][0] = imgcoord[cnt][0];
            linecoord[*linecnt][1] = imgcoord[cnt][1];
            (*linecnt)++;
        }
    }

/* Sort based on X coord of centroid */
for(i=0;i< (*linecnt);i++)
    for(j=(*linecnt - 1);j>i;j--)
        if(linecoord[j][dir] <= linecoord[j-1][dir])
        {
            tempx = linecoord[j][0];
            tempy = linecoord[j][1];
            linecoord[j][0] = linecoord[j-1][0];
            linecoord[j][1] = linecoord[j-1][1];
            linecoord[j-1][0] = tempx;
            linecoord[j-1][1] = tempy;
        }
}
```

```
newmomt.c      Fri Jul 27 09:55:59 1990      1
/* Standard vxWorks include file.          */
/* Standard I/O library include file.      */
#include "vxWorks.h"
#include "stdioLib.h"
#include "memLib.h"
#include "cal.h"

momt(picin,frame,item1,newnumobj,newnumhole)
    short *frame,*newnumobj,*newnumhole;
    unsigned char *picin;
    MOMENTS item1[200];
{
    static MOMENTS item[200];
    static int map[2000];
    int i,j,labela,labelb,labelc,nlabel,labelobj,labelhole,index;
    int numobj=0,numhole=0,noise=32;
    int off=1000;
    int temp1, temp2, temp3, temp4, temp5, temp6;
    int *label;
    int l,iadd,iup,curitm,newitm,off2=100;
    int obj,newobj,counter;
    double a1,a2,asp=0.85;

/* It is necessary to use CALLOC since label must be initialized to zero */
    if((label=(int *) calloc((unsigned) (IMGX*IMGY),
                           (unsigned) sizeof(int))) == NULL)
        logMsg("NO MEMORY FOR LABEL\n");

    labelobj=0; labelhole=0;
    *(map+off)=0;

    for(j=(*frame+1); j<(*frame+1)); j++)
    {
        for(i=(*frame+2)+1; i<(*frame+3); i++)
        {
            index=i+IMGX*j;
            if(*(picin+index) == 0)
            {
                if(*(label+index-1) <= 0)
                {
                    if(*(label+index-IMGX) <= 0)
                    {
                        if(*(label+index-(IMGX+1)) <= 0)
                        {
                            if(*(label+index-(IMGX-1)) <= 0)
                            {
                                labelobj++;
                                *(label+index)=labelobj;
                                *(map+off+labelobj)=labelobj;
                            }
                            else
                                *(label+index)=*(label+index-(IMGX-1));
                        }
                    }
                }
                else
                {
                    if(*(label+index-(IMGX-1)) <= 0)
                        *(label+index)=*(label+index-(IMGX+1));
                    else
                    {
                        labela=*(label+index-(IMGX+1));
                        labelb=*(label+index-(IMGX-1));
                        if(labela == labelb)
                            *(label+index)=labela;
                        else
                        {
                            if(*(map+off+labela)>*(map+off+labelb))

```

```
        labelc= *(map+off+labelb);
    else
        labelc= *(map+off+labela);
    *(label+index)=labelc;
    *(map+off+ *(map+off+labela))=labelc;
    *(map+off+ *(map+off+labelb))=labelc;
    for(nlabel=1; nlabel < (labelobj+1);
        nlabel++)
    {
        *(map+off+nlabel)= *(map+off +
            *(map+off+nlabel));
    }
}
}
}
}
else
    *(label+index)= *(label+index-IMGX);
}
else
{
    if(*(label+index-(IMGX-1)) > 0)
    {
        labela= *(label+index-1);
        labelb= *(label+index-(IMGX-1));
        if(labela == labelb)
            *(label+index)=labela;
        else
        {
            if(*(map+off+labela)> *(map+off+labelb))
                labelc= *(map+off+labelb);
            else
                labelc= *(map+off+labela);
            *(label+index)=labelc;
            *(map+off+ *(map+off+labela))=labelc;
            *(map+off+ *(map+off+labelb))=labelc;
            for(nlabel=1; nlabel < (labelobj+1); nlabel++)
            {
                *(map+off+nlabel)= *(map+off+ *(map+off+nlabel));
            }
        }
    }
    else
        *(label+index)= *(label+index-1);
}
}
else
{
    if(*(label+index-1) > 0)
    {
        if(*(label+index-IMGX) > 0)
        {
            labelhole--;
            *(label+index)=labelhole;
            *(map+off+labelhole)=labelhole;
        }
        else
            *(label+index)= *(label+index-IMGX);
    }
}
else
{
    if(*(label+index-IMGX) > 0)
        *(label+index)= *(label+index-1);
    else
```

```
{  
    labela= *(label+index-IMGX);  
    labelb= *(label+index-1);  
    if(labela == labelb)  
        *(label+index)=labela;  
    else  
    {  
        if(*(map+off+labela)> *(map+off+labelb))  
            labelc= *(map+off+labela);  
        else  
            labelc= *(map+off+labelb);  
        *(label+index)=labelc;  
        *(map+off+ *(map+off+labela))=labelc;  
        *(map+off+ *(map+off+labelb))=labelc;  
        for(nlabel= 0; nlabel > (labelhole+1); nlabel--)  
        {  
            *(map+off+nlabel)= *(map+off+ *(map+off+nlabel));  
        }  
    }  
}  
}  
  
for(nlabel=1; nlabel < (labelobj+1); nlabel++)  
{  
    if(*(map+off+nlabel) == nlabel)  
    {  
        numobj++;  
        *(map+off+nlabel) = numobj;  
        templ = numobj+off2;  
        item[templ].m00 = 0.0;  
        item[templ].m10 = 0.0;  
        item[templ].m01 = 0.0;  
        item[templ].m20 = 0.0;  
        item[templ].m02 = 0.0;  
        item[templ].m11 = 0.0;  
        item[templ].inter = 0.0;  
        item[templ].xmin = IMGX;  
        item[templ].ymin = IMGY;  
        item[templ].xmax = 0;  
        item[templ].ymax = 0;  
        item[templ].rot = 0.0;  
    }  
    else  
    {  
        *(map+off+nlabel) = *(map+off+ *(map+off+nlabel));  
    }  
}  
  
for(nlabel=(0); nlabel > (labelhole-1); nlabel--)  
{  
    if(*(map+off+nlabel) == nlabel)  
    {  
        *(map+off+nlabel)=numhole;  
        temp2 = off2+numhole;  
        item[temp2].m00=0.0;  
        item[temp2].m10=0.0;  
        item[temp2].m01=0.0;  
        item[temp2].m20=0.0;  
        item[temp2].m02=0.0;
```

```
    item[temp2].m11=0.0;
    item[temp2].inter=0.0;
    item[temp2].xmin=IMGX;
    item[temp2].ymin=IMGY;
    item[temp2].xmax=0;
    item[temp2].ymax=0;
    item[temp2].rot=0.0;
    numhole--;
}
else
{
    *(map+off+nlabel)= *(map+off+ * (map+off+nlabel));
}
}

numhole= -1*(numhole+1);
*newnumhole = numhole;

for(j= (*frame+1); j< (*frame+1)); j++)
{
    curitm= *(map+off+ * (label+j*IMGX));
    iup= *(frame+2);
    for(i= (*frame+2)+1; i< (*frame+3)); i++)
    {
        index=i+j*IMGX;
        newitm= *(map+off+ * (label+index));
        temp3 = curitm+off2;
        temp4 = newitm+off2;
        if (newitm != curitm)
        {
            l=i-iup;
            item[temp3].m00=item[temp3].m00+l;
            iadd=(l*(2*iup+(l-1))/2;
            item[temp3].m10=item[temp3].m10+iadd;
            item[temp3].m01=item[temp3].m01+l*j;
            item[temp3].m11=item[temp3].m11+iadd*j;
            item[temp3].m20=item[temp3].m20+
                (l*(6*iup*(iup+(l-1))+(l-1)*(2*l-1)))/6;
            item[temp3].m02=item[temp3].m02+l*j*j;
            item[temp4].xmin= fncref(item[temp4].xmin,i);
            item[temp4].ymin= fncref(item[temp4].ymin,j);
            item[temp3].xmax= fncref(item[temp3].xmax,(i-1));
            item[temp3].ymax= fncref(item[temp3].ymax,j);
            iup=i;
        }
        else
        {
            if(newitm > 0)
            {
                if(((*label+index+1)>0)&&(*label+index+IMGX)>0)&&
                    ((*label+index-IMGX)>0))
                    item[temp4].inter=item[temp4].inter +1;
            }
        }
        curitm=newitm;
    }
}

counter = 0;
*newnumobj = 0;
for(obj=1; obj<=numobj; obj++)
{
    temp5 = obj+off2;
    if((item[temp5].m00) > noise)
```

```
{  
    *newnumobj = ++counter;  
    newobj = *newnumobj;  
    temp6 = newobj+off2;  
    item1[temp6].m00=item[temp5].m00;  
    item1[temp6].gravx=item[temp5].m10/item[temp5].m00;  
    item1[temp6].gravy=item[temp5].m01/item[temp5].m00;  
    item1[temp6].m20=(item[temp5].m20/item[temp5].m00-  
        (item1[temp6].gravx*item1[temp6].gravx));  
    item1[temp6].m02=(item[temp5].m02/item[temp5].m00-  
        (item1[temp6].gravy*item1[temp6].gravy))*asp*asp;  
    item1[temp6].m11=(item[temp5].m11/item[temp5].m00-  
        (item1[temp6].gravx*item1[temp6].gravy))*asp;  
    item1[temp6].perim=item[temp5].m00-item[temp5].inter;  
    a1=2.0*item1[temp6].m11;  
    a2=item1[temp6].m02-item1[temp6].m20;  
    item1[temp6].rot= -(PI/2.0) - 0.5*atan2(a1,a2);  
    item1[temp6].m01=item[temp5].m01;  
    item1[temp6].m10=item[temp5].m10;  
    item1[temp6].inter=item[temp5].inter;  
    item1[temp6].xmin=item[temp5].xmin;  
    item1[temp6].ymin=item[temp5].ymin;  
    item1[temp6].xmax=item[temp5].xmax;  
    item1[temp6].ymax=item[temp5].ymax;  
}  
}  
free(label);  
}
```

```
*****
* This procedure reads in camera calibration parameters that have been *
* determined already. The file that is being read must be in the          *
* following format:                                              *
*                                                               *
* camer1-rot[0].....camer1-rot[8]                                     *
* camer1-trans[0].....camer1-trans[2]                                    *
* camer1-dist[0].....camer1-dist[1]                                     *
* camer1-focal length                                                 *
* camer1-scale factor                                                 *
* camera2-rot[0].....camera2-rot[8]                                     *
* camera2-trans[0].....camera2-trans[2]                                    *
* camera2-dist[0].....camera2-dist[1]                                     *
* camera2-focal length                                                 *
* camera2-scale factor                                                 *
*                                                               *
* All parameters are of type double and their should be no empty lines   *
* in the file. Use the save_params routine to write parameters to a      *
* file.                                                       *
*****/
```

```
#include "cal.h"
#include "ext.h"

int read_params(caml,cam2)
CAM_PARAMS *caml, *cam2;

{
    char fname[20];
    FILE *infile;

    printf("Input the name of the file from which to read the parameters.\n");
    printf("The name must be less than 20 characters.\n");
    scanf(" %[^\n]",fname);

    if((infile = fopen(fname, "r")) != NULL)
    {

        /* Read the first camera parameters. */                                */
        fscanf(infile, " %lf %lf %lf %lf %lf %lf %lf %lf\n",
               &(caml->rot[0]),&(caml->rot[1]),&(caml->rot[2]),
               &(caml->rot[3]),&(caml->rot[4]),&(caml->rot[5]),
               &(caml->rot[6]),&(caml->rot[7]),&(caml->rot[8]));
        fscanf(infile, " %lf %lf %lf\n",
               &(caml->trans[0]),&(caml->trans[1]),&(caml->trans[2]));
        fscanf(infile, " %lf %lf\n",&(caml->dist[0]),&(caml->dist[1]));
        fscanf(infile, " %lf\n",&(caml->focal));
        fscanf(infile, " %lf\n",&(caml->scale));

        /* Read the second camera parameters. */                                */
        fscanf(infile, " %lf %lf %lf %lf %lf %lf %lf %lf\n",
               &(cam2->rot[0]),&(cam2->rot[1]),&(cam2->rot[2]),
               &(cam2->rot[3]),&(cam2->rot[4]),&(cam2->rot[5]),
               &(cam2->rot[6]),&(cam2->rot[7]),&(cam2->rot[8]));
        fscanf(infile, " %lf %lf %lf\n",
               &(cam2->trans[0]),&(cam2->trans[1]),&(cam2->trans[2]));
        fscanf(infile, " %lf %lf\n",&(cam2->dist[0]),&(cam2->dist[1]));
        fscanf(infile, " %lf\n",&(cam2->focal));
        fscanf(infile, " %lf\n",&(cam2->scale));

        fclose(infile);
    }

    else
    {

```

```
    printf("Can not open the file %s\n", fname);
    return(-1);
}

return(0);
}
```

```
*****
* This procedure saves the camara calibration parameters to a user *
* specified file. The information will be placed in the file using *
* the following format: *
*
* camer1-rot[0].....cameral-rot[8] *
* camer1-trans[0].....cameral-trans[2] *
* camer1-dist[0].....cameral-dist[1] *
* camer1-focal length *
* camer1-scale factor *
* camera2-rot[0].....camera2-rot[8] *
* camera2-trans[0].....camera2-trans[2] *
* camera2-dist[0].....camera2-dist[1] *
* camera2-focal length *
* camera2-scale factor *
*
* All parameters are of type double and their is no empty lines *
* in the file. Use the read_params routine to read the parameters *
* from a file. *
*****
#include "cal.h"
#include "ext.h"

int save_params(cam1,cam2)
CAM_PARAMS *cam1, *cam2;

{
char fname[20];
FILE *outfile;

printf("Enter the name of the file in which the parameters will be saved.\n");
printf("The name must be less than 20 characters.\n");
scanf(" %[^\n]",fname);

if((outfile = fopen(fname, "w")) != NULL)
{
/* Read the first camera parameters. */
fprintf(outfile, " %lf %lf %lf %lf %lf %lf %lf\n",
(cam1->rot[0]),(cam1->rot[1]),(cam1->rot[2]),
(cam1->rot[3]),(cam1->rot[4]),(cam1->rot[5]),
(cam1->rot[6]),(cam1->rot[7]),(cam1->rot[8]));
fprintf(outfile, " %lf %lf\n",
(cam1->trans[0]),(cam1->trans[1]),(cam1->trans[2]));
fprintf(outfile, " %lf\n", (cam1->dist[0]),(cam1->dist[1]));
fprintf(outfile, " %lf\n", (cam1->focal));
fprintf(outfile, " %lf\n", (cam1->scale));

/* Read the second camera parameters. */
fprintf(outfile, " %lf %lf %lf %lf %lf %lf %lf\n",
(cam2->rot[0]),(cam2->rot[1]),(cam2->rot[2]),
(cam2->rot[3]),(cam2->rot[4]),(cam2->rot[5]),
(cam2->rot[6]),(cam2->rot[7]),(cam2->rot[8]));
fprintf(outfile, " %lf %lf\n",
(cam2->trans[0]),(cam2->trans[1]),(cam2->trans[2]));
fprintf(outfile, " %lf\n", (cam2->dist[0]),(cam2->dist[1]));
fprintf(outfile, " %lf\n", (cam2->focal));
fprintf(outfile, " %lf\n", (cam2->scale));

fclose(outfile);
}

else
{
```

save_params.c Mon Nov 20 13:31:28 1989 2

```
    printf("Can not open the file %s\n", fname);
    return(-1);
}

return(0);
}
```

```
#include "cal.h"

svbksb(u,w,v,m,n,b,x)
double u[],w[],v[],b[],x[];
int m,n;
{
    int i,j,jj;
    double s,*tmp;
    tmp=(double *)malloc(m*sizeof(double));
    for(j=0;j<n;++j)
    {
        s=0.0;
        if(w[j] != 0.0)
        {
            for(i=0;i<m;++i)
                s+=u[i*n+j]*b[i];
            s/=w[j];
        }
        tmp[j]=s;
    }
    for(j=0;j<n;++j)
    {
        s=0.0;
        for(jj=0;jj<n;++jj)
            s+=v[j*n+jj]*tmp[jj];
        x[j]=s;
    }
    free(tmp); /* don't forget to free */
} /* end of function svbksb() */
```

svdcmp.c Sat Nov 18 18:21:49 1989 1

```
#include "cal.h"

svdcmp(a,m,n,w,v)
double a[],w[],v[];
int m,n;

{
    int i,j,k,l;
    double f,g,h,scale,s,anorm;
    double rv1[100];
    int its,nm,jj;
    double c,x,y,z;

    g = 0.0;
    scale = 0.0;
    anorm = 0.0;
    if (m < n)
        printf("You must augment A with extra rows\n");
    for(i=0;i<n;i++) /* 25 */
        l = i + 1;
        rv1[i] = scale * g;
        g = 0.0;
        s = 0.0;
        scale = 0.0;
        if (i < (m-1)) {
            for (k=i;k<m;k++) /* 11 */
                scale = scale + fabs(a[k*n+i]);
            /* 11 */
            if (scale != 0.0) {
                for(k=i;k<m;k++) /* 12 */
                    a[k*n+i] = a[k*n+i]/scale;
                s = s + a[k*n+i]*a[k*n+i];
                /* 12 */
                f = a[i*n+i];
                g = -copysign(sqrt(s),f);
                h = f*g-s;
                a[i*n+i] = f-g;
                if (i != (n-1)) {
                    for(j=l;j<n;j++) /* 15 */
                        s = 0.0;
                    for(k=i;k<m;k++) /* 13 */
                        s = s + a[k*n+i]*a[k*n+j];
                    /* 13 */
                    f = s/h;
                    for(k=i;k<m;k++) /* 14 */
                        a[k*n+j] = a[k*n+j] + f*a[k*n+i];
                    /* 14 */
                    /* 15 */
                }
                /* 16 */
                a[k*n+i] = scale * a[k*n+i];
                /* 16 */
            }
            /* endif */
        }
        /* endif */

    w[i] = scale * g;
    g = 0.0;
    s = 0.0;
    scale = 0.0;
    if ((i<m) && (i != (n-1))) {
        for(k=l;k<n;k++) /* 17 */
            scale = scale + fabs(a[i*n+k]);
        /* 17 */
    }
}
```

```
if (scale != 0.0) {
    for(k=1;k<n;k++) { /* 18 */
        a[i*n+k] = a[i*n+k] / scale;
        s = s + a[i*n+k]*a[i*n+k];
    } /* 18 */
    f = a[i*n+1];
    g = - copysign(sqrt(s),f);
    h = f*g-s;
    a[i*n+1] = f-g;
    for(k=1;k<n;k++) { /* 19 */
        rv1[k] = a[i*n+k]/h;
    } /* 19 */
    if (i != (m-1)) {
        for(j=1;j<m;j++) { /* 23 */
            s = 0.0;
            for(k=1;k<n;k++) { /* 21 */
                s = s + a[j*n+k]*a[i*n+k];
            } /* 21 */
            for(k=1;k<n;k++) { /* 22 */
                a[j*n+k] = a[j*n+k] + s*rv1[k];
            } /* 22 */
        } /* 23 */
        /* endif */
    }
    for(k=1;k<n;k++) { /* 24 */
        a[i*n+k] = scale*a[i*n+k];
    } /* 24 */
} /* endif */
/* endif */
anorm = fnormmax(anorm, (fabs(w[i])+fabs(rv1[i])));
} /* 25 */

for(i=(n-1);i>=0;i--) { /* 32 */
    if (i < (n-1)) {
        if (g != 0.0) {
            for(j=1;j<n;j++) { /* 26 */
                v[j*n+i] =(a[i*n+j] / a[i*n+1]) / g;
            } /* 26 */
            for(j=1;j<n;j++) { /* 29 */
                s = 0.0;
                for(k=1;k<n;k++) { /* 27 */
                    s = s + a[i*n+k] * v[k*n+j];
                } /* 27 */
                for(k=1;k<n;k++) { /* 28 */
                    v[k*n+j] = v[k*n+j] + s * v[k*n+i];
                } /* 28 */
            } /* 29 */
            /* endif */
        }
        for(j=1;j<n;j++) { /* 31 */
            v[i*n+j] = 0.0;
            v[j*n+i] = 0.0;
        } /* 31 */
        /* endif */
    }
    v[i*n+i] = 1.0;
    g = rv1[i];
    l = i;
}
} /* 32 */

for(i=(n-1);i>=0;i--) { /* 39 */
    l = i + 1;
    g = w[i];
    if (i < (n-1)) {
        for(j=1;j<n;j++) { /* 33 */
            a[i*n+j] = 0.0;
        } /* 33 */
    }
}
```

```
        } /* endif */
    if ( g != 0.0) {
        g = 1.0 / g;
        if ( i != (n-1)) {
            for(j=1;j<n;j++) { /* 36 */
                s = 0.0;
                for(k=1;k<m;k++) { /* 34 */
                    s = s + a[k*n+i] * a[k*n+j];
                } /* 34 */
                f = ( s / a[i*n+i]) * g;
                for(k=i;k<m;k++) { /* 35 */
                    a[k*n+j] = a[k*n+j] + f * a[k*n+i];
                } /* 35 */
                /* 36 */
            } /* endif */
        } /* 37 */
        for(j=i;j<m;j++) { /* 37 */
            a[j*n+i] = a[j*n+i] * g;
        } /* 37 */
    } /* endif */
else
    for(j=i;j<m;j++) { /* 38 */
        a[j*n+i] = 0.0;
    } /* 38 */
a[i*n+i] = a[i*n+i] + 1.0; /* 39 */

for(k=(n-1);k>=0;k--) { /* 49 */
    for(its=1;its<=30;its++) { /* 48 */
        for(l=k;l>=0;l--) { /* 41 */
            nm = l - 1;
            if ( (fabs(rv1[l]) + anorm) == anorm) break;
            if ( (fabs(w[nm]) + anorm) == anorm) break;
        } /* 41 */
        if ( (fabs(rv1[l]) + anorm) != anorm)
        {
            c = 0.0;
            s = 1.0;
            for(i=l;i<k;i++) { /* 43 */
                f = s * rv1[i];
                if ((fabs(f) + anorm) != anorm) {
                    g = w[i];
                    h = sqrt(f*f+g*g);
                    w[i] = h;
                    h = 1.0 / h;
                    c = g * h;
                    s = -(f * h);
                    for(j=0;j<m;j++) { /* 42 */
                        y = a[j*n+nm];
                        z = a[j*n+i];
                        a[j*n+nm] = (y * c) + (z * s);
                        a[j*n+i] = -(y * s) + (z * c);
                    } /* 42 */
                } /* endif */
            } /* 43 */
        } /* end of my if */
    z = w[k];
    if (l == k) {
        if (z < 0.0) {
            w[k] = -z;
            for(j=0;j<n;j++) { /* 44 */
                v[j*n+k] = -v[j*n+k];
            } /* 44 */
        } /* endif */
    } /* 44 */
} /* endif */
```

```
    continue;
}
if (its == 30)
    printf(" No convergence in 30 iterations\n");

x = w[l];
nm = k - 1;
y = w[nm];
g = rvl[nm];
h = rvl[k];
f = ((y-z) * (y+z) + (g-h) * (g+h)) / (2.0*h*y);
g = sqrt(f*f+1.0);
f = ((x-z) * (x+z) + h * ((y / (f + copysign(g,f))) - h)) / x;
c = 1.0;
s = 1.0;
for(j=1;j<=nm;j++) {           /* 47 */
    i = j+1;
    g = rvl[i];
    y = w[i];
    h = s * g;
    g = c * g;
    z = sqrt(f*f + h*h);
    rvl[j] = z;
    c = f/z;
    s = h/z;
    f = (x*c) + (g*s);
    g = -(x*s) + (g*c);
    h = y * s;
    y = y * c;
    for(jj=0;jj<n;jj++) {      /* 45 */
        x = v[jj*n+j];
        z = v[jj*n+i];
        v[jj*n+j] = (x*c) + (z*s);
        v[jj*n+i] = -(x*s) + (z*c);
    }                           /* 45 */
    z = sqrt(f*f + h*h);
    w[j] = z;
    if (z != 0.0) {
        z = 1.0/z;
        c = f * z;
        s = h * z;
    }                           /* endif */
    f = (c*g) + (s*y);
    x = -(s*g) + (c*y);
    for(jj=0;jj<m;jj++) {     /* 46 */
        y = a[jj*n+j];
        z = a[jj*n+i];
        a[jj*n+j] = (y*c) + (z*s);
        a[jj*n+i] = -(y*s) + (z*c);
    }                           /* 46 */
}                           /* 47 */
rvl[l] = 0.0;
rvl[k] = f;
w[k] = x;
}                           /* 48 */
}                           /* 49 */
}
```

where.c Sat Nov 18 18:21:49 1989 1

```
#include "puma.h"

where(vector)
float vector[6];
{
struct location compound;
int superror;

compound.quality="location";
superror = 1;
while (superror != 0)
{
    start_supervisor(&superror);
    if(superror != 0)
        printf("Bad Superror\n");
    else
        printf("Supervisor Enabled.\n");
}
here_v(&compound,"wait",&superror);
if(superror == 0)
    decompose_v(vector,&compound);
else
    printf("code superror=%d\n",superror);
abort_supervisor(&superror);
}
```